Paper accepted
34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)
San Diego, CA, November 11-15 2019

Empirical Study of Programming to an Interface

Benoît Verhaeghe*, Christopher Fuhrman†, Latifa Guerrouj†, Nicolas Anquetil‡ and Stéphane Ducasse§
*Berger-Levrault, France
†Dept. of Software Engineering and IT, École de technologie supérieure, Montreal, Canada
‡Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL, F-59000 Lille, France
§RMoD - Inria Nord Europe, Lille, France
*{firstname}.{lastname}@berger-levrault.com †{firstname}.{lastname}@etsmtl.ca ‡§{firstname}.{lastname}@inria.fr

***Abstract*—A popular recommendation to programmers in object-oriented software is to "program to an interface, not an implementation" (PTI). Expected benefits include increased simplicity from abstraction, decreased dependency on implementations, and higher flexibility. Yet, interfaces must be immutable, excessive class hierarchies can be a form of complexity, and "speculative generality" is a known code smell. To advance the empirical knowledge of PTI, we conducted an empirical investigation that involves 126 Java projects on GitHub, aiming to measuring the decreased dependency benefits (in terms of cochange).**

***Index Terms*—Java interfaces, coupling, empirical study, software repositories, cochange, GitHub**

I. Introduction

Object-oriented software design has leveraged many important principles from modular design, including abstraction (*e.g.,* standardized interfaces), and information hiding. Java [1] provides a mechanism for abstraction known as an *interface*, which "typically declares one or more abstract methods; otherwise unrelated classes can implement the interface by providing implementations for its abstract methods." Like its analog in C♯, it can be a solution to the limitation of single inheritance. C++ virtual functions are a similar abstraction.

A well known, object-oriented heuristic [2] states it is better to "program to an interface, not an implementation." In other words, programming to an interface (PTI) means that if a client `C` needs to use `X` for some service, it is better for `C` to use an abstraction (`IX`, which could also be the superclass) of `X` providing the service. This heuristic is based on common sense and experience, but empirical evidence is lacking on whether its use is always beneficial.

Expected benefits of PTI include simplicity, flexibility, and decreased dependency. The API of `IX` will likely have fewer methods than `X`. It can be *published* [3], meaning `IX` should be immutable and client programmers can depend on it. This allows developers of `X` to change it within the constraints of its published API.

On the other hand, this use of abstractions can have other consequences. It can add levels to a hierarchy, and has been shown as detracting from understandability in certain models [4]. Abstracting a service with an interface when there is but one implementation is a smell called "Speculative Generality" [5]. Sometimes interfaces are wrong and cannot remain immutable, affecting not only clients but all implementations. Better empirical knowledge of the PTI principle could enable automated recommendation systems for developers regarding the consequences of applying this heuristic.

Because of the multiple consequences of the PTI principle, many approaches could be taken in studying them, including controlled experiments with human subjects, and natural experiments with data from software artifacts and histories produced by humans. Some aspects of PTI, e.g., simplicity, might best be studied with a controlled human subject experiment. Yet, these experiments have complexities: human subjects (software developers) are in high demand today, experiments must be approved by university research boards, and human subjects cannot simply forget their experiences once they have participated in an experiment. Thus controlled experiments with real developers are difficult to realize and are constrained to a waterfall process. Although natural experiments have their own limitations (the context in which their data are obtained, the validation of measures, etc.), they can be automated (and easily repeatable) on large, publicly available sets of data.

In this article we present an original natural experiment to find evidence supporting PTI with respect to one of its expected benefits: decreasing dependencies. We perform a natural experiment using archival data from histories of 126 popular open-source Java systems on GitHub. Using a static-analysis model based on previous work from Rufiange and Fuhrman [6], we identify relationships between clients of implementations that use (or do not use) abstractions to access them. We formulate and test a hypothesis on how likely these two kinds of accesses are to be involved in changes, and find that both cases resulted in relatively low frequencies (<10%) of cochange.

In the next section we present the background on PTI. Section III presents the methodology of our empirical study aiming at measuring benefits of decreased dependencies. Section IV discusses the operationalization details, while Section V reveals and discusses the results. We address the threats to validity in Section VI, while we present the related work in Section VII. Finally, Section VIII concludes the paper with a summary of the findings and directions for future work.

II. Programming to an Interface

Interfaces can be used as abstractions to achieve information hiding [7]. Larman [8] coined this as *Protected Variation* and argued it is equivalent to the Open-Closed Principle [9]:

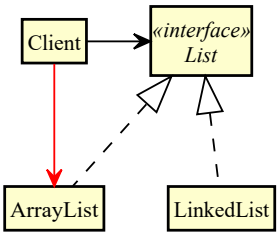

Figure 1: Clients can access implementations directly (ArrayList) or only through an interface (List)

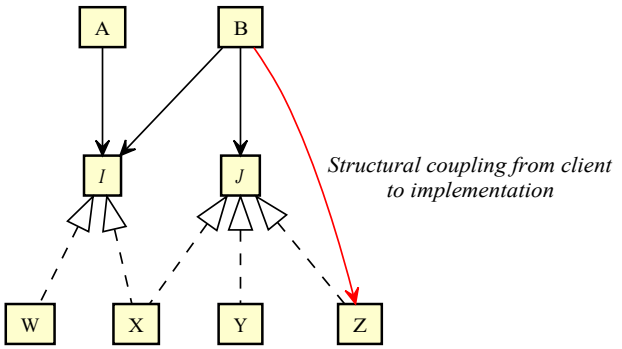

Figure 2: Model for classifying coupling with interfaces. Clients A and B use interfaces I and J with implementations W, X, Y and Z. This example yields the sets $P = \{(A \to W)(A \to X)(B \to W)(B \to X)(B \to Y)\}$ and $U = \{(B \to Z)\}$.

modules should be both open (for extension and adaptation) and closed (to avoid modification that affect clients). Protected Variation uses the term *interface* in a general sense of a stable abstraction, and predates the concept of a Java `interface`. Java `interfaces`, however, can be easily identified in source code, and their intention as design abstractions is generally straightforward.

PTI intends for client classes to use an interface abstraction rather than its concrete (implementing) classes. In Java, for example, the `Client` of the `List` interface in Figure 1 can use an `ArrayList` directly, or it can treat it as an implementation of `List`, without knowing that it's an `ArrayList`. Advantages to this include that the client's code should be simpler, since it only sees the methods of `List` rather than all the methods of `ArrayList`. There is greater flexibility for `Client` to later use the `LinkedList` implementation for performance reasons. Developers of implementations can modify their code without breaking the client. However, interfaces must be immutable; otherwise changes will be required in clients and implementations.

Instances have to be created, and this can only come from calls to constructors in concrete classes (implementations). An instruction in Java such as `List l = new ArrayList();` is called *upcasting*, an essential mechanism in PTI. Although it results in dependency on the implementation through the constructor, these accesses can be isolated via dependency injection [10] or factories. However, this has been shown to harm understandability [11]. Fowler [5] proposes a related refactoring: "Replace Constructor with Factory Function".

III. Experiment Setup

As mentioned in the introduction, different kinds of experiments could be done to measure how the consequences of PTI manifest themselves. Rather than run a controlled experiment with human subjects, we present a novel empirical investigation that focuses on analyzing the evolution of easily accessible data from artifacts produced by developers. The experiment looks at whether structural characteristics of following PTI has measurable benefits in terms of decreased dependencies. Following the notion that adhering to PTI should isolate clients from dependencies on the implementation they are using, we frame our experiment in a context of cochange pairs (*e.g.,* Geipel and Schweitzer [12]). However, we limit our study of pairs to the context of Java interfaces, where a cochange pair consists of a client and an implementation of a given interface, because interfaces are easily identifiable in Java source code. If the reduced-dependency benefit of PTI holds, we expect that client-implementation pairs with *direct dependency* should be linked by cochange significantly more often than those pairs with *indirect dependencies*. Thus we propose the following research question:

RQ: *Are interface clients that are directly coupled to an implementation more likely subject to cochange than interface clients which are potentially indirectly coupled to implementations?*

$\boldsymbol{H_0}$: *There is no significant difference between the cochange frequency of directly coupled client-implementation pairs and indirectly coupled client-implementation pairs.*

For the sake of simplicity, we perform our analyses at a file-level granularity (*i.e.,* `.java` files) rather than methods or variables.

A. Dependencies

Figure 2 shows how clients of an interface can have structural coupling to implementations of the interface (*e.g.,* client `B` $\to$ implementation `Z`), or how they can be structurally independent of implementations (*e.g.,* client `A` and implementation `W`). Since using interfaces without coupling between clients and implementations of the interface is called *protected variation* [8], we distinguish client-implementation pairs as either *protected* (having no direct structural coupling) or *unprotected* (having direct structural coupling).

The *protected pair set*, P, contains all pairs of clients c and implementations m where c depends on an interface i of which m is an implementation, and c does not structurally depend on m. Client `A` (Figure 2) could use implementations `W` or `X` via a factory or dependency injection, resulting in the protected pairs $(A \to W)$ and $(A \to X)$.

Conversely, the *unprotected pair set*, U, contains all pairs of clients c and implementations m, where c has some direct structural dependency on m and on an interface i implemented by m. Client `B` refers to both `J` and `Z` modeled as the unprotected pair $(B \rightarrow Z)$.

In both sets class c depends explicitly on the interface i implemented by m. The sets differ only by the fact that c also depends directly on m (U) or not (P). Thus, we ignore dependencies between a client of a class where an interface exists that the client does not depend on, even if the client only uses methods declared by the interface. Some additional exclusions are applied:

- Interfaces and implementations *not* defined inside the project (*e.g.,* `java.util.Iterator`).
- Interfaces that do not declare any methods (*i.e., tagging* or *marking* interfaces).
- Anonymous classes (because they don't correspond to unique files).

B. Cochange

As in previous work [12], [13], cochange is extracted from project change events, namely all classes whose changes have been committed by the same author at the same time. Specifically, the following were ignored:

- *add* events, as opposed to *change*, as the former do not provide evolvability information for the system [12].
- revisions with over 10 files [14] to reduce the impact of changes that are not related to functional aspects of source code, *e.g.,* updating licence information in all classes.
- revisions where interface updates have been committed, because interface changes imply implementation and possibly client cochanges.

Modern version control systems can contain multiple branches [15]. To avoid the complexity of analyzing merges to determine unique change events, we consider change events on a single, deterministic path through the repository. We start at the HEAD commit of a repository and walk backwards in history, always choosing the first parent at merge commits, until there are no more commits in the project history.

We define a history set H containing directional pairs of classes coming from change events. Since a change event containing classes A and B could imply $A \rightarrow B$ and $B \rightarrow A$, we add both possibilities to H, similar to the approach done by Ajienka and Capiluppi [13].

The *Cochanged Protected Dependencies ratio* (CPD) is the percentage of protected client-implementation pairs that have cochanged, and *Cochanged Unprotected Dependencies ratio* (CUD) is the percentage of unprotected client-implementation pairs that have cochanged: $CPD(\%) = \frac{|H \cap P|}{|P|}$ and $CUD(\%) = \frac{|H \cap U|}{|U|}$.

IV. Experiment Operationalization

A GraphQL query on GitHub (2019-03-27) yielded the most popular projects with Java source code, sorted by the sum of the number of stars and forks. Projects with less than 95% Java content were excluded, leaving 140 projects. From this we excluded 4 projects that declared no interfaces, and 9 projects that had 10 or fewer commits, and one project `janishar/mit-deep-learning-book-pdf` with a single Java file, resulting in a total of 126 projects for the study.

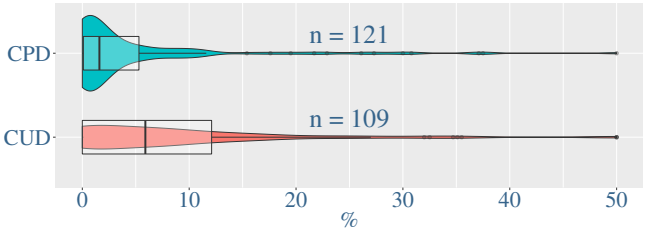

Figure 3: Box/Violin plots for ratios of Cochanged Protected Dependencies (CPD) and Cochanged Unprotected Dependencies (CUD). Projects where P or U are empty are excluded.

VerveineJ generated a FAMIX Java model of the latest revision of each project. Moose [16] determined the project's interfaces, clients and implementations. Protected and unprotected pairs were determined by querying the model.

For cochange information, we mined each project by walking the commit history of each project using `libgit2` and Pharo Iceberg, filtering commits per the constraints describe above. We generated pairs of Java classes that appear in commits to compute H, P, U, CPD and CUD as described above. Data files (comma separated value format) were generated for each step in the analysis to provide transparency and to allow validation and failure recovery of the steps.

V. Results

Figure 3 shows the CUD and CPD distributions for all projects. We see that protected cochange pairs have a lower ratio of occurring compared to unprotected cochange pairs. We calculated the statistical significance of these results using Wilcoxon signed rank test with Holm adjustment to adjust the p-value. In addition, we computed the effect size using the non-parametric Cliff's delta. We obtained $p < 0.001$ and $d = 0.32$ (small) respectively, which allowed us to reject our null-hypothesis H_0. The effect size is, however, small.

For most projects the cochange ratio is small (under 10%). This result differs from others [12] where mean cochange probabilities of more than 30% were reported. We only considered client-implementation pairs, whereas the former study looked at all structurally coupled pairs. Client-implementation pairs appear to be a smaller source of cochange within the scope of a project. One explanation could be that because clients bind to interfaces, which are normally stable elements in a design, they are more stable with respect to implementations regardless of how the clients access them. The medians of the numbers of protected pairs $|P|$ and unprotected pairs $|U|$ are 227.5 and 36 respectively, showing that most projects have many more protected pairs than unprotected pairs.

VI. Threats to validity

Regarding internal validity, one threat is that cochange history is mined from only the *first* branch after a merge; useful cochange history could be missed on ignored branches.

Concerning external validity, the results should be applicable to all Java projects. However, they are limited to the scope of the sample of (popular) projects we found on GitHub. To mitigate this threat, we studied a large ($n = 126$) sample of open-source projects.

The threats of the operational aspects (mining online repositories, age of pairs, sampling of only Java open-source projects) mentioned in related work [12], [13] apply also to this experiment and are not repeated here due to space limitations.

VII. Related work

The relationship of Java clients to interfaces and their implementations was explored by Rufiange and Fuhrman [6], although no empirical results were done. Geipel and Schweitzer did an empirical study [12] to look at how structural coupling and cochange were related. They used project histories from 35 projects under CVS and found empirical evidence that change propagates along paths of dependency. Their results showed that cochange could not be inferred by only looking at dependencies. Ajienka and Capiluppi [13] looked at structural coupling and so-called *logical* coupling, which they mined using the Apriori algorithm from commit histories of open-source Java projects. They concluded that not all cochanged pairs of classes are linked by structural dependencies, but structurally coupled pairs of classes usually include logical dependencies. Our study is effectively a subset of classes (clients, interfaces, and their implementations) in a similar context.

Abdeen, Sahraoui *et al.,* [17] proposed metrics for assessing interface design, including the "Program to Interface Principle" (PTIP), and examined empirically for one version only of the project (no history) whether interfaces in real-world applications respect the PTIP. They defined a metric called *Loose Program To Interface* (LPTI), ranging from 0 to 1, where the latter signifies strong adherence to PTIP. With respect to our model, $LPTI\,(i) = 1$ implies that the interface i has only protected pairs.

Sabané, Guéhéneuc, *et al.,* [18] did an empirical study examining the impact of the Fragile Base-class problem and found no correlation with change- and fault-proneness. They suggest an explanation that the popularity of dependency-injection, which favors avoiding the problem by favoring composition over inheritance. New implementations created this way would be clients in our model, and who likely access *directly* concrete implementations (via delegation), modeled as unprotected pairs.

VIII. Conclusion and future work

We suggest a novel empirical approach to study the principle known as programming to an interface (PTI), specifically targeting the expected benefits in terms of decreased dependencies. A class-pair model keeps track of cochange in the context of the structural coupling of clients and implementations of a given interface. Our results on a set of 126 open-source Java projects show that the directly coupled client-implementation (unprotected) pairs are less likely to be involved in cochanges than their indirectly coupled (protected) counterparts.

As future work, we intend to extend our approach of a natural experiment with finer detail classification of pairs (*e.g.,* microarchitectures, factories, dependency injection, delegation, « creates » coupling and even method-to-method pairs), to better understand the implications of PTI.

More research is needed to address the other aspects of PTI, such as flexibility, simplicity, protecting internals and even working around single inheritance limitations in Java/C♯. Qualitative studies as well as mining studies can investigate those aspects.

References

[1] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[3] M. Fowler, "Public versus published interfaces," *IEEE Software*, vol. 19, no. 2, pp. 18–19, March 2002.

[4] J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Nov. 2018.

[6] S. Rufiange and C. P. Fuhrman, "Visualizing protected variations in evolving software designs," *Journal of Systems and Software*, vol. 88, pp. 231–249, 2014.

[7] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *CACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[8] C. Larman, "Protected variation: The importance of being closed," *Software, IEEE*, vol. 18, no. 3, pp. 89–91, 2001.

[9] B. Meyer, *Object-oriented Software Construction*. Prentice-Hall, 1988.

[10] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004. [Online]. Available: https://martinfowler.com/articles/injection.html

[11] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in API design: A usability evaluation," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 302–312.

[12] M. M. Geipel and F. Schweitzer, "The link between dependency and cochange: Empirical evidence," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1432–1444, Nov 2012.

[13] N. Ajienka and A. Capiluppi, "Understanding the interplay between the logical and structural coupling of software classes," *Journal of Systems and Software*, vol. 134, pp. 120–137, 2017.

[14] H. Kagdi, M. Gethers, and D. Poshyvanyk, "Integrating conceptual and logical couplings for change impact analysis in software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 933–969, Oct 2013.

[15] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10.

[16] S. Ducasse, N. Anquetil, U. Bhatti, A. Cavalcante Hora, J. Laval, and T. Girba, "MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family," RMod – INRIA Lille-Nord Europe, Tech. Rep., 2011. [Online]. Available: http://rmod.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf

[17] H. Abdeen, H. Sahraoui, and O. Shata, "How we design interfaces, and how to assess it," in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013, pp. 80–89.

[18] A. Sabané, Y. Guéhéneuc, V. Arnaoudova, and G. Antoniol, "Fragile base-class problem, problem?" *Empirical Software Engineering*, vol. 22, no. 5, pp. 2612–2657, 2017.